SANDIA REPORT

SAND2013-0725
Unlimited Release
Printed 01/2013

Exomerge User's Manual: A lightweight Python interface for manipulating Exodus files

Timothy D. Kostka

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-0725
Unlimited Release
Printed 01/2013

Exomerge User's Manual: A lightweight Python interface for manipulating Exodus files

Timothy D. Kostka
Multi-physics Modeling and Simulation
Sandia National Laboratories
P.O. Box 969
Livermore, CA 94550-0969
tdkostk@sandia.gov

Abstract

Exomerge is a lightweight Python module for reading, manipulating and writing data within ExodusII files. It is built upon a Python wrapper around the ExodusII API functions. This module, the Python wrapper, and the ExodusII libraries are available as part of the standard SIERRA installation.

Acknowledgments

The author appreciates the efforts of Dave Littlewood, Tim Shelton and Mike Veilleux for the creation of the Python wrapper around the ExodusII API, without which this software would not be possible.

The author appreciates the help of Jay Dike, Jay Foulk and Mike Veilleux for reviewing and improving this document.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Contents

Chapter 1

Introduction

Exomerge is a lightweight Python module for accessing and manipulating information within ExodusII files. In particular, Exomerge is not simply a wrapper for the ExodusII API library. Rather, it has functionality to simplify the process of manipulating files to avoid limitations of the ExodusII file format.

Originally, we needed the capability to selectively merge elements blocks from multiple database files. A simple Python script was devised to merge these blocks, hence the name Exo*merge*. Since then, the module has expanded as new capabilities were required. In its current form, Exomerge provides an easy way to interface with an ExodusII file and modify nearly everything therein.

Chapter 2

Getting Started

All of the tools needed to use Exomerge, including Python, are available within the standard SIERRA distribution. After setting up the required environment variable (see next section), we recommend new users consult the example scripts in the back of this manual to see some example uses.

Setting up the environment

While the Exomerge module is included within the SIERRA distribution, the path to the module is not automatically available. One must set the `PYTHONPATH` environment variable to the location of the `exomerge.py` file. This can be done by exporting the variable within the Bash shell and then running Python normally.

```
$ export PYTHONPATH=/path/to/exomerge.py
$ python
```

Alternatively, one can set the path and call python in one step. This will only set the environment variable for that particular command.

```
$ PYTHONPATH=/path/to/exomerge.py python
```

Once this is set up, the `import exomerge` Python command will load the module.

Executing scripts

The typical usage scenario is to write a script which performs some action and then call Python using the script as an argument. For example, the following script will print the version number.

```
# ---- script.py ----
import exomerge
# print the version number
print "This is Exomerge v%g." % (exomerge.VERSION)
```

To run the script, one would type the following on the command line:

```
$ python script.py
This is Exomerge v7.0.
```

Running interactively

One can also run Python interactively, which may be useful for debugging purposes. For example, the following will bring up the documentation for the `delete_element_block` function.

```
$ python
>>> import exomerge
>>> # create an empty model
>>> model = exomerge.ExodusModel()
>>> # bring up help
>>> help(model.delete_element_block)
Help on method delete_element_block in module exomerge:

delete_element_block(self, element_block_ids) method of exomerge.ExodusModel instance
    Delete one or more element blocks.

    Examples:
    >>> model.delete_element_block(1)
    >>> model.delete_element_block([1, 3, 4])
```

On-the-fly documentation

This manual lists commands available within Exomerge at the time of writing. If new commands are added, or if any current commands have changed, one can access documentation through Python's pydoc feature in the console.

```
$ # bring up all documentation
$ pydoc exomerge
$ # get help on a particular function
$ pydoc exomerge.ExodusModel.delete_element_block
```

Documentation obtained in this manner is guaranteed to be accurate, at least in syntax, as it is based off of the exact Exomerge file you are using.

A very quick introduction to Python

The Python language is very easy to learn and to use in particular because it does not require a compiler. Or rather, when something is compiled, it does so behind the scenes in a way that is invisible to the user.

As a convention, the line prefix `">>>"` is used as the prompt in Python while the prefix `"..."` is used a continuation prompt. These conventions are followed in this manual in examples which need to differentiate between input and output. Line prefixes should not appear in any scripts.

To print a simple string, the syntax is very easy. Strings in python may use either single or double quotes.

```
>>> # This is a comment.
>>> print 'Hello world!'
Hello world!
```

To print more complex output, the `"%"` mark may be used in a `printf`-style format.

```
>>> x = 2
>>> y = x + 3
>>> print 'Currently, x=%g and y=%g.' % (x, y)
Currently, x=2 and y=5.
```

Lists objects can be easily manipulated.

```
>>> x = [1, 2, 3]
>>> x.append(4)
>>> print x
[1, 2, 3, 4]
>>> print x[0]
1
>>> print x[-1]
4
```

Standard `if/then` constructs exist for conditional statements. By convention, nested statements within Python are indented by exactly 4 spaces.

```
>>> x = 5
>>> if x == 5:
...     print 'x is 5'
>>> else:
...     print 'x is not 5'
x is 5
```

One may loop over objects using a `for` loop.

```
>>> for i in [0, 1, 2]:
...     print i
0
1
2
```

For more advanced examples, please consult the internet.

Chapter 3

Basics

This chapter provides a quick overview of the ExodusII file format and general usage information for Exomerge.

Anatomy of an ExodusII file

Because Exomerge is built as a tool for manipulating ExodusII files, it is important to understand the information inside a file. The ExodusII webpage[1] lists the following description for the file format.

> “EXODUS II is a model developed to store and retrieve transient data for finite element analyses. It is used for preprocessing, postprocessing, as well as code to code data transfer.”

Within a given ExodusII file, the following entities are present.

- **nodes**

  A node is simply a location in three-dimensional space.

- **element blocks**

  An element block is a collection of elements of the same type. For example, you cannot have hexahedral and tetrahedral elements in the same block.

- **elements**

  Each element block in the model can contain any number of elements. Each element is defined by the list of nodes which form their connectivity list.

- **side sets**

  A side set is a collection of element faces. Internally, these are stored by listing a pair of the form `(element_index, face_index)`.

- **node sets**

  A node set is a collection of nodes.

[1] http://sourceforge.net/projects/exodusii/

- **timesteps**

  A file may have any number of timesteps, including zero. Field values are defined for each timestep.

- **global variables**

  A global variable has a single value for each timestep.

- **node fields**

  A node field lists the value of the field at each node for each timestep.

- **element fields**

  An element field lists a the value of the field at each element within a given block for each timestep. Element fields may not be defined on every element block, but if they are defined, a value is defined for each element within that block.

- **side set fields**

  Similar to element fields, a side set field is a list of values for each member of the side set for each timestep. Side set fields may not be defined for all side sets.

- **node set fields**

  Similar to element fields, a node set field is a list of values for each member of the node set for each timestep. Node set fields may not be defined for all node sets.

- **information records**

  The information records are a list of strings within the file. Most commonly, they are used to store the input deck within the results file, but they can be used for any purpose.

- **quality assurance information**

  A list of quality assurance field may be defined in the file. These hold the names and version numbers of each program used to create or modify the file.

Methodology of Exomerge

With ExodusII, one must know the exact number of nodes, element blocks, side sets, and other entities before a file can be created. Once a file exists, these numbers cannot change. This makes the procedure of, for instance, deleting an element block from a file a non-trivial problem.

Exomerge gets around this limitation by holding objects in memory and only writing to a file when requested. This allows us to easily perform operations such as merging files, deleting timesteps, and creating new fields to name just a few.

This also imposes a fundamental limitation on the size of a model which may be imported based on the available RAM. Although this is not an issue in most situations, it may be an issue for very large files. To mitigate this, one can load only the relevant portion of the output file, such as a particular element block or a particular timestep, to cut down on the memory cost.

Missing features

While efforts have been made to make Exomerge fully compatible with ExodusII, some features have not yet been implemented. Some of these are listed below.

- **Node id maps and element id maps**

  Instead of numbering each node from 1 to N, where N is the number of nodes, it is possible within ExodusII to define a node map which gives each node an index independent of its internal storage index. Similarly, an element map may also be defined. This feature was not implemented into Exomerge.

- **Element attributes**

  As opposed to element fields which are defined at each element for each timestep, element attributes are defined once for each element, independent of the number of timesteps. This feature is not widely used and is not supported within Exomerge.

- **Object names**

  Within ExodusII, it is possible to assign a name to an element block and refer to it by that name instead of the element block number. This feature is not widely used and is not supported within Exomerge. Any object names which exist are lost when the file is loaded.

Entity specification

When using Exomerge functions, one often has to refer to entities such as element blocks.

Entities within a model fall into one of three categories.

- **Named entities**

  Named entities are referred to by a string which gives their name. This includes global variables, node fields, side set fields, and node set fields.

- **Integer entities**

  Integer entities are referred to by an integer known as their id. This includes element blocks, node sets, and side sets.

- **Timesteps**

  Timesteps are stored as floating-point numbers and must be specified by their exact value. If a timestep of 3.01 exists and you refer to timestep 3.0, it will not be found.

Special identifiers

The following special identifiers may be used.

- `'none'`

  This will refer to no entities.

- `'all'`

  This will refer to all possible entities.

- `'first'`

  This will refer to the first entity. For named objects, entities are sorted alphabetically except in the case of multi-component quantities such as tensors. For numeric entities, entities are sorted ascendingly. If no objects exist, an error is thrown.

- `'last'`

  This will refer to the last entity. For named objects, entities are sorted alphabetically except in the case of multi-component quantities such as tensors. For numeric entities, entities are sorted ascendingly. If no objects exist, an error is thrown.

- `'auto'`

  In the special case in which exactly one entity is present, this will refer to it. For example, if only one timestep is present, it may be specified by its value or by `'auto'`. If zero or more than one is present, an error is thrown.

- **Wildcard (`*`)**

  The asterisk wildcard character (`*`) may be used to match any number of characters for entities referred to by name, such as element fields. For example, if one wanted to load all stress components, one would specify `element_field_names='stress_*'` in the argument to `import_model()`. Note that it is possible for zero matches to occur.

For example, the following three commands will produce identical results.

```
model.delete_node_fields('*')
model.delete_node_fields('all')
model.delete_node_fields(model.get_node_field_names())
```

Supported element types

For the vast majority of operations, including reading and writing files, all element types are supported.

For functions which require element topology information to be known, such as `export_stl_file()` or `create_side_set_from_expression()`, the following element types are supported.

- Hexahedrons (8-node, 20-node)
- Tetrahedrons (4-node, 10-node)
- Wedges (6-node, 15-node, 16-node)
- Quadrilaterals (4-node, 8-node)

- Triangles (3-node, 6-node)
- Lines/Beams (2-node, 3-node)
- Points (1-node)

Case sensitivity

In general, strings are case sensitive. That means trying to access `'temp'` when only `'TEMP'` exists will produce an error. To help mitigate this issue, one can use the `to_lowercase()` method to convert all entity names to lowercase.

Warning and error messages

In the case in which a function fails, every effort has been made to provide helpful information as to why the error occurred. For example, trying to delete an element block that does not exist produce the following warning message.

```
$ python
>>> import exomerge
>>> # create an empty model
>>> model = exomerge.ExodusModel()
>>> # try to delete a non-existant element block
>>> model.delete_element_block(1)

###############################################################################
#                                                                             #
#  WARNING: Reference to undefined element block.                             #
#                                                                             #
#  Message: A reference to element block "1" was encountered but is not       #
#           defined.  This element block will not be included in the          #
#           operation.                                                        #
#                                                                             #
#           There are 0 defined element blocks:                               #
#                                                                             #
###############################################################################
```

In this case, a warning message was produced as opposed to an error message because the program could continue. If the program can not continue, an error message will instead be output and the program will exit.

If the program crashes with a Python-style stack trace as opposed to a formatted error message, it suggests the problem is elsewhere.

Chapter 4

Code quality checks

While it is not expected that users will run code quality checks, they are listed here to (a) let users know what automated checks are in place and (b) to document the process for future developers. When the software is updated, these checks should be done before release.

We use three types of tests to verify the quality of the code. Style checks verify that the source code is written in a standard format. Syntax checks verify the source code is valid. Unit tests verify that the code actually works. While only unit tests are important to the end user, style and syntax checks are useful for developers.

Syntax/style checks

We use three tools to check for errors in syntax and/or style.

- pep8 - http://pypi.python.org/pypi/pep8

  This program checks for compliance with the standard Python coding style guide, PEP8.

- pyflakes - https://launchpad.net/pyflakes

  This program checks for syntax errors.

- pylint - http://www.pylint.org/

  This program does many checks for both syntax and style and can occasionally be more useful than the other two programs. Its output, however, is rather verbose.

To execute any of these checks, with the appropriate program installed, execute any of the following.

```
$ pep8 /path/to/exomerge.py
$ pyflakes /path/to/exomerge.py
$ pylint /path/to/exomerge.py
```

The first two checks should yield zero output if there are no errors. The last program, pylint, is much more verbose.

Unit tests

This chapter describes the unit tests done to improve software quality and catch potential bugs. The unit test script `exomerge_unit_test.py` is located in the regression test suite of SIERRA and is

not typically available to users.

Unit tests are run automatically on a nightly basis to help ensure the quality of the code.

Methodology

Instead of creating a custom unit test for each and every function, we instead perform a randomized test of the function on a model in which each unit test can modify the model in some manner. In this way, we hope to catch otherwise difficult to spot bugs which happen only in corner cases.

Each public function within Exomerge must have a corresponding function within the unit test module and must be named accordingly. For example, the `_test_rename_node_field()` must test the `rename_node_field()` public function.

Example unit test

For each unit test, we attempt to modify the model by using the appropriate function in some way. For example, the following unit tests attempts to rename a randomly chosen node field from those present. If none are available, it returns `False` to communicate the test was unable to be done.

```
    def _test_rename_node_field(self):
        name = self._random_node_field_name()
        if name is None:
            return False
        self.model.rename_node_field(name, self._new_node_field_name())
```

Example output

The output will list the number and name of each unit test called.

```
$ python exomerge_unit_test.py 1000
We found 97 unit tests and 97 public functions.
Opening exodus file: exomerge_unit_test.e
Closing exodus file: exomerge_unit_test.e
[1] _test_calculate_node_field
[2] _test_calculate_node_field
[3] _test_delete_timestep
...
[998] _test_add_faces_to_side_set
[999] _test_create_interpolated_timestep
[1000] _test_displace_element_blocks

Ran 1000 tests in 17.6181 seconds.

Success
```

If a failure occurs, it will be output and the program will exit with a non-zero exit status.

Chapter 5

API Documentation

This chapter provides documentation for each public function within Exomerge. All of these functions are part of the `exomerge.ExodusModel` object and use the typical object-oriented function calling form.

add_faces_to_side_set

Add the given faces to the side set.

The parameter `new_side_set_members` should be a list of tuples of the form:

- `(element_block_id, local_element_index, element_side_index)`

Interface

```
def add_faces_to_side_set(self, side_set_id, new_side_set_members):
```

Parameters

`side_set_id`	A side set id.
`new_side_set_members`	A list of side set members.

Example

```
>>> model.add_faces_to_side_set(1, [(2, 0, 3), (3, 0, 4)])
```

add_nodes_to_node_set

Add nodes to an existing node set.

Interface

```
def add_nodes_to_node_set(self, node_set_id, new_node_set_members):
```

Parameters

`node_set_id`	A node set id.
`new_node_set_members`	A list of node indices.

Example

```
>>> model.add_nodes_to_node_set(1, [4, 5, 6, 7])
```

calculate_element_centroids

Calculate and store the centroid of each element.

This will approximate the element centroid as the nodal average of each element and will store that value in an element field. Since a timestep must be defined in order for element fields to exist, one will be created if none exist.

By default, the centroid will be stored in the fields `centroid_x`, `centroid_y`, and `centroid_z`.

Interface

```
def calculate_element_centroids(self,
                                element_field_name_prefix='centroid_',
                                element_block_ids='all'):
```

Parameters

`element_field_name_prefix`	A prefix for an element field.
`element_block_ids`	A list of element block ids.

Example

```
>>> model.calculate_element_centroids()
```

calculate_element_field

Calculate a new element field from the given expression.

The expression may include the following variables:

- `time` to refer to the current time
- global variables (by name)
- element fields (by name)

Interface

```
def calculate_element_field(self, expression, element_block_ids='all'):
```

Parameters

`expression`	An expression to evaluate.
`element_block_ids`	A list of element block ids.

Example

```
>>> model.calculate_element_field('pressure = (stress_xx + '
...                               'stress_yy + stress_zz) / -3')
```

calculate_element_field_maximum

For each element field listed, calculate the maximum value and store it as a global variable.

To also store the element block id which contains the element with this value, set `calculate_block_id=True`.

To also store the location of the centroid of the element with this value, set `calculate_location=True`.

Interface

```
def calculate_element_field_maximum(self,
                                    element_field_names,
                                    element_block_ids='all',
                                    calculate_location=False,
                                    calculate_block_id=False):
```

Parameters

`element_field_names`	A list of element field names.
`element_block_ids`	A list of element block ids.
`calculate_location`	If `True`, the location of this value will be calculated and stored.
`calculate_block_id`	If `True`, the element block containing this value will be calculated and stored.

Example

```
>>> model.calculate_element_field_maximum('eqps')
```

calculate_element_field_minimum

For each element field listed, calculate the minimum value and store it as a global variable.

To also store the element block id which contains the element with this value, set `calculate_block_id=True`.

To also store the location of the centroid of the element with this value, set `calculate_location=True`.

Interface

```
def calculate_element_field_minimum(self,
                                    element_field_names,
                                    element_block_ids='all',
                                    calculate_location=False,
                                    calculate_block_id=False):
```

Parameters

`element_field_names`	A list of element field names.
`element_block_ids`	A list of element block ids.
`calculate_location`	If `True`, the location of this value will be calculated and stored.
`calculate_block_id`	If `True`, the element block containing this value will be calculated and stored.

Example

```
>>> model.calculate_element_field_minimum('eqps')
```

calculate_global_variable

Calculate a new global variable from the given expression.

The expression may include the following variables:

- `time` to refer to the current time
- global variables (by name)

Interface

```
def calculate_global_variable(self, expression):
```

Parameters

`expression`	An expression to evaluate.

Example

```
>>> model.calculate_global_variable('time_squared = time ^ 2')
>>> model.calculate_global_variable('total = potential + kinetic')
```

calculate_node_field

Calculate a new node field from the given expression.

The expression may include the following variables:

- `time` to refer to the current time
- global variables (by name)
- node fields (by name)
- model coordinates (`x`, `y`, and `z`)

Interface

```
def calculate_node_field(self, expression):
```

Parameters

`expression` An expression to evaluate.

Example

```
>>> model.calculate_node_field('temp_C = temp_K - 273.15')
```

calculate_node_field_maximum

For each node field listed, calculate the maximum value and store it as a global variable.

To also store the location of the node with this value, set `calculate_location=True`.

Interface

```
def calculate_node_field_maximum(self,
                                 node_field_names,
                                 element_block_ids='auto',
                                 calculate_location=False):
```

Parameters

`node_field_names` A list of node field names.

`element_block_ids` A list of element block ids.

`calculate_location` If `True`, the location of this value will be calculated and stored.

Example

```
>>> model.calculate_node_field_maximum('temp')
>>> model.global_variables['temp_max']
```

calculate_node_field_minimum

For each node field listed, calculate the minimum value and store it as a global variable.

By default, all nodes are used.

To also store the location of the node with this value, set `calculate_location=True`.

Interface

```
def calculate_node_field_minimum(self,
                                 node_field_names,
                                 element_block_ids='auto',
                                 calculate_location=False):
```

Parameters

`node_field_names`	A list of node field names.
`element_block_ids`	A list of element block ids.
`calculate_location`	If `True`, the location of this value will be calculated and stored.

Example

```
>>> model.calculate_node_field_minimum('temp')
>>> model.global_variables['temp_min']
```

calculate_node_set_field

Calculate a new node set field from the given expression.

The expression may include the following variables:

- `time` to refer to the current time
- global variables (by name)
- node fields (by name)
- node set fields (by name)
- model coordinates (`X`, `Y`, and `Z`)

Interface

```
def calculate_node_set_field(self, expression, node_set_ids='all'):
```

Parameters

`expression`	An expression to evaluate.
`node_set_ids`	A list of node set ids.

Example

```
>>> model.calculate_node_set_field('temp_C = temp_K - 273.15')
```

calculate_side_set_field

Calculate a new side set field from the given expression.

The expression may include the following variables:

- `time` to refer to the current time
- global variables (by name)

Interface

```
def calculate_side_set_field(self, expression, side_set_ids='all'):
```

Parameters

`expression`	An expression to evaluate.
`side_set_ids`	A list of side set ids.

Example

```
>>> model.calculate_side_set_field('temp_C = temp_K - 273.15')
```

combine_element_blocks

Combine multiple element blocks into a single block.

By default, the target element block id will be the smallest of the merged element block ids.

The element blocks to combine must have the same element type.

Interface

```
def combine_element_blocks(self,
                           element_block_ids,
                           target_element_block_id='auto'):
```

Parameters

`element_block_ids`	A list of element block ids.
`target_element_block_id`	An element block id.

Example

```
>>> model.combine_element_blocks('all', 1)
```

convert_element_field_to_node_field

Convert an element field to a node field by performing an element average.

For each node, the value of the field at that node will be the average of the values in every element which shares that node for elements on which the field is defined.

Interface

```
def convert_element_field_to_node_field(self,
                                        element_field_name,
                                        node_field_name='auto'):
```

Parameters

`element_field_name`	An element field name.
`node_field_name`	A node field name.

Example

```
>>> model.convert_element_field_to_node_field('temperature')
```

convert_hex8_block_to_tet4_block

Convert a block of `hex8` elements to a block of `tet4` elements using the given conversion scheme.

Side sets are updated accordingly.

Node sets are not modified.

Currently, only the `hex24tet` scheme is implemented, which creates 24 `tet4` element for eeach `hex8` element in the original mesh.

Interface

```
def convert_hex8_block_to_tet4_block(self,
                                     element_block_id,
                                     scheme='hex24tet'):
```

Parameters

`element_block_id`	An element block id.
`scheme`	A name of the scheme.

Example

```
>>> model.convert_hex8_block_to_tet4_block(1)
```

convert_node_field_to_element_field

Convert a node field to an element field by performing a nodal average.

For each element, the value of the field at that element will be the average of the values in each node of that element.

Interface

```
def convert_node_field_to_element_field(self,
                                        node_field_name,
                                        element_field_name='auto',
                                        element_block_ids='all'):
```

Parameters

`node_field_name`	A node field name.
`element_field_name`	An element field name.
`element_block_ids`	A list of element block ids.

Example

```
>>> model.convert_node_field_to_element_field('temperature')
```

convert_side_set_to_cohesive_zone

Convert the given side sets into a block of cohesive zone elements.

The given side set must contain faces which are internal to the body and shared by exactly two `hex8` elements. Support for other element types is not implemented.

Interface

```
def convert_side_set_to_cohesive_zone(self,
                                      side_set_ids,
                                      new_element_block_id):
```

Parameters

`side_set_ids`	A list of side set ids.
`new_element_block_id`	An element block id.

Example

```
>>> model.convert_side_set_to_cohesive_zone(1, 2)
```

copy_timestep

Create a copy of an existing timestep.

This copies all information from the old timestep including values of global variables, node fields, element fields, node set fields and side set fields.

Interface

```
def copy_timestep(self, current_timestep, new_timestep):
```

Parameters

`current_timestep`	A timestep value.
`new_timestep`	A timestep value.

Example

```
object.copy_timestep(0.0, 1.0)
```

create_averaged_element_field

Create an element field by averaging the given field values.

Interface

```
def create_averaged_element_field(self,
                                  from_element_field_names,
                                  new_element_field_name,
                                  element_block_ids='all'):
```

Parameters

`from_element_field_names`	A list of element field names.
`new_element_field_name`	An element field name.
`element_block_ids`	A list of element block ids.

Example

```
>>> model.create_averaged_element_field(['temp_1', 'temp_2'],
...                                     'temp_avg')
>>> model.create_averaged_element_field('temp_*', 'temp_avg')
```

create_displacement_field

Create the displacement field if it doesn't already exist and assign it a value of zero everywhere.

Interface

```
def create_displacement_field(self):
```

No parameters

Example

```
>>> model.create_displacement_field()
```

create_element_block

Create a new element block.

The nodes for the elements in the block must have already been defined.

The info list should be comprised of the following information.

- `[element_type, element_count, nodes_per_element, 0]`

For example, the following would be valid.

- `["hex8", elements, 8, 0]`

The connectivity list should be a shallow list of element connectivity and must be of length `element_count * nodes_per_element`.

Interface

```
def create_element_block(self, element_block_id, info, connectivity=None):
```

Parameters

`element_block_id` An element block id.

`info`

`connectivity` A list of node indices defining the element connectivity.

Example

```
>>> model.create_element_block(1, ['hex8', 0, 8, 0])
```

create_element_field

Create an element field on the given element blocks and assign it a default value.

Interface

```
def create_element_field(self,
                         element_field_name,
                         element_block_ids='all',
                         value='auto'):
```

Parameters

`element_field_name`	An element field name.
`element_block_ids`	A list of element block ids.
`value`	A value.

Example

```
>>> model.create_element_field('eqps', 0.0)
```

create_global_variable

Create a new global variable.

Interface

```
def create_global_variable(self, global_variable_name, value='auto'):
```

Parameters

`global_variable_name`	A global variable name.
`value`	A value.

Example

```
>>> model.create_global_variable('gravitational_acceleration', 9.8)
```

create_interpolated_timestep

Create a new timestep by interpolating the solution of neighboring steps.

This does not extrapolate, so the given timestep to be interpolated must lie within the range of timesteps already defined.

By default, `cubic` interpolation is used. This can be changed to `linear` interpolation if desired.

Interface

```
def create_interpolated_timestep(self, timestep, interpolation='cubic'):
```

Parameters

`timestep` A timestep value

`interpolation`

Example

```
>>> model.create_interpolated_timestep(0.5)
>>> model.create_interpolated_timestep(0.5, interpolation='linear')
```

create_node_field

Create a node field and assign it a default value.

Interface

```
def create_node_field(self, node_field_name, value='auto'):
```

Parameters

`node_field_name` A node field name.

`value` A value.

Example

```
>>> model.create_node_field('temperature', 298.15)
```

create_node_set

Create a node side from the given list of node indices.

Interface

```
def create_node_set(self, node_set_id, node_set_members=None):
```

Parameters

`node_set_id`	A node set id.
`node_set_members`	A list of node indices.

Example

```
>>> model.create_node_set(1, [0, 1, 2, 3])
```

create_node_set_field

Create a node set field on the given node sets and assign it a default value.

Interface

```
def create_node_set_field(self,
                          node_set_field_name,
                          node_set_ids='all',
                          value='auto'):
```

Parameters

`node_set_field_name`	A node set field name.
`node_set_ids`	A list of node set ids.
`value`	A value.

Example

```
>>> model.create_node_set_field('temperature', 13, 298.15)
```

create_node_set_from_side_set

Create a node set containing all nodes used by faces in the given side set.

Interface

```
def create_node_set_from_side_set(self, node_set_id, side_set_id):
```

Parameters

`node_set_id` A node set id.

`side_set_id` A side set id.

Example

```
>>> model.create_node_set_from_side_set(1, 1)
```

create_nodes

Create nodes corresponding to the given coordinate list.

The list must contain coordinate triples `[x, y, z]` defined for each new node. The new nodes are assigned a local index starting with the current length of the nodes list.

Interface

```
def create_nodes(self, new_nodes):
```

Parameters

`new_nodes` A list of node indices.

Example

```
>>> model.create_nodes([[0.0, 0.0, 0.0], [1.0, 2.0, 3.0]])
```

create_side_set

Create a side set from the given element faces.

If the side set already exists, the given faces will be added.

side_set_members should be a list of tuples of the form:

- `(element_block_id, local_element_index, element_side_index)`

Interface

```
def create_side_set(self, side_set_id, side_set_members=None):
```

Parameters

`side_set_id`	A side set id.
`side_set_members`	A list of element faces.

Example

```
>>> model.create_side_set(1, [(1, 0, 1), (1, 0, 2)])
```

create_side_set_field

Create a side set field on the given side sets and assign it a default value.

Interface

```
def create_side_set_field(self,
                          side_set_field_name,
                          side_set_ids='all',
                          value='auto'):
```

Parameters

`side_set_field_name`	A side set field name.
`side_set_ids`	A list of side set ids.
`value`	A value.

Example

```
>>> model.create_side_set_field('temperature', 13, 298.15)
```

create_side_set_from_expression

Create a side side from external element faces which satisfy the given expression.

For example, if the model had a symmetry plane, `Y == 0` would select all element faces on this plane.

Interface

```
def create_side_set_from_expression(self,
                                    side_set_id,
                                    expression,
                                    element_block_ids='all',
                                    tolerance='auto',
                                    timesteps='last_if_any',
                                    zero_member_warning=True):
```

Parameters

`side_set_id`	A side set id.
`expression`	An expression to evaluate.
`element_block_ids`	A list of element block ids.
`tolerance`	A tolerance value.
`timesteps`	A list of any number of timesteps.
`zero_member_warning`	If `True` and the list evaluates to zero members, output a warning.

Example

```
>>> model.create_side_set_from_expression(1, 'Y == 0')
```

create_timestep

Create a new timestep.

Field information at the created timestep is set to the default value.

Interface

```
def create_timestep(self, timestep):
```

Parameters

`timestep`	A timestep value

Example

```
>>> model.create_timestep(0)
```

delete_element_block

Delete one or more element blocks.

This function may also be called by its plural form `delete_element_blocks()`.

Interface

```
def delete_element_block(self, element_block_ids):
```

Parameters

`element_block_ids` A list of element block ids.

Example

```
>>> model.delete_element_block(1)
>>> model.delete_element_block([1, 3, 4])
```

delete_element_field

Delete one or more element fields.

This function may also be called by its plural form `delete_element_fields()`.

Interface

```
def delete_element_field(self, element_field_names, element_block_ids='all'):
```

Parameters

`element_field_names` A list of element field names.

`element_block_ids` A list of element block ids.

Example

```
>>> model.delete_element_field('eqps')
>>> model.delete_element_field('all')
```

delete_empty_node_sets

Delete all side sets with zero members.

Interface

```
def delete_empty_node_sets(self):
```

No parameters

Example

```
>>> model.delete_empty_node_sets()
```

delete_empty_side_sets

Delete all side sets with zero members.

Interface

```
def delete_empty_side_sets(self):
```

No parameters

Example

```
>>> model.delete_empty_side_sets()
```

delete_global_variable

Delete one or more global variables.

This function may also be called by its plural form `delete_global_variables()`.

Interface

```
def delete_global_variable(self, global_variable_names):
```

Parameters

`global_variable_names` A list of global variable names.

Example

```
>>> model.delete_global_variable('internal_energy')
>>> model.delete_global_variable('all')
```

delete_node_field

Delete one or more node fields.

This function may also be called by its plural form `delete_node_fields()`.

Interface

```
def delete_node_field(self, node_field_names):
```

Parameters

`node_field_names` A list of node field names.

Example

```
>>> model.delete_node_field('temperature')
>>> model.delete_node_field('all')
>>> model.delete_node_field('disp_*')
```

delete_node_set

Delete the given node set.

This function may also be called by its plural form `delete_node_sets()`.

Interface

```
def delete_node_set(self, node_set_ids):
```

Parameters

`node_set_ids` A list of node set ids.

Example

```
>>> model.delete_node_set(1)
```

delete_node_set_field

Delete one or more node set fields.

This function may also be called by its plural form `delete_node_set_fields()`.

Interface

```
def delete_node_set_field(self, node_set_field_names, node_set_ids='all'):
```

Parameters

`node_set_field_names`	A list of node set field names.
`node_set_ids`	A list of node set ids.

Example

```
>>> model.delete_node_set_field('contact_pressure', 1)
```

delete_side_set

Delete one or more side sets.

This function may also be called by its plural form `delete_side_sets()`.

Interface

```
def delete_side_set(self, side_set_ids):
```

Parameters

`side_set_ids`	A list of side set ids.

Example

```
>>> model.delete_side_set(1)
>>> model.delete_side_set('all')
```

delete_side_set_field

Delete one or more side set fields.

This function may also be called by its plural form `delete_side_set_fields()`.

Interface

```
def delete_side_set_field(self, side_set_field_names, side_set_ids='all'):
```

Parameters

`side_set_field_names`	A list of side set field names.
`side_set_ids`	A list of side set ids.

Example

```
>>> model.delete_side_set_field('contact_pressure', 1)
```

delete_timestep

Delete one or more timesteps.

Because fields are defined on each timestep, this also deletes field values for the corresponding timestep.

This function may also be called by its plural form `delete_timesteps()`.

Interface

```
def delete_timestep(self, timesteps):
```

Parameters

`timesteps`	A list of any number of timesteps.

Example

```
>>> model.delete_timestep(0.0)
>>> model.delete_timestep('all')
```

delete_unused_nodes

Delete nodes which are not used by any elements.

Interface

```
def delete_unused_nodes(self):
```

No parameters

Example

```
>>> model.delete_unused_nodes()
```

displace_element_blocks

Displace all nodes in the given element blocks.

This function operates on the displacement field rather than the model coordinate field. To operate on the model coordinates, use `translate_element_blocks()`.

Interface

```
def displace_element_blocks(self,
                            element_block_ids,
                            vector,
                            timesteps='all',
                            check_for_merged_nodes=True):
```

Parameters

`element_block_ids`	A list of element block ids.
`vector`	A vector.
`timesteps`	A list of any number of timesteps.
`check_for_merged_nodes`	If `True`, check for merged nodes between element blocks to avoid inconsistent output.

Example

```
>>> model.displace_element_blocks(1, [1.0, 2.0, 3.0])
>>> model.displace_element_blocks('all', [1.0, 2.0, 3.0])
```

displacement_field_exists

Return `True` if a displacement field exists.

Interface

```
def displacement_field_exists(self):
```

No parameters

Example

```
>>> model.displacement_field_exists()
```

duplicate_element_block

Create an duplicate of the given element block.

Nodes are duplicated. The new element block references these duplicated nodes, not the original ones.

Interface

```
def duplicate_element_block(self, old_element_block_id, new_element_block_id):
```

Parameters

`old_element_block_id` An element block id.

`new_element_block_id` An element block id.

Example

```
>>> model.duplicate_element_block(1, 2)
```

element_block_exists

Return `True` if the given element block exists.

Interface

```
def element_block_exists(self, element_block_id):
```

Parameters

`element_block_id` An element block id.

Example

```
>>> model.element_block_exists(1)
```

element_field_exists

Return `True` if the given element field exists on the given element blocks.

Interface

```
def element_field_exists(self, element_field_name, element_block_ids='all'):
```

Parameters

`element_field_name` An element field name.

`element_block_ids` A list of element block ids.

Example

```
>>> model.element_field_exists('eqps')
```

export

This is a helper function which calls the appropriate exporter based on extension of the filename given.

Arguments are passed on through.

The following extensions will call the appropriate functions:

- `WRL --> export_wrl_model`
- `STL --> export_stl_file`
- `E, G, EXO --> export_model`

Interface

```
def export(self, filename, *args, **kwargs):
```

Parameters

`filename` A filename.

Example

```
>>> model.export('result.e')
>>> model.export('result.stl')
>>> model.export('result.wrl', 'eqps')
```

export_model

Write out the current model to an ExodusII file.

Interface

```
def export_model(self,
                 filename='output_exomerge.e',
                 element_block_ids='all',
                 timesteps='all',
                 side_set_ids='all',
                 node_set_ids='all',
                 global_variable_names='auto',
                 node_field_names='auto',
                 element_field_names='auto',
                 side_set_field_names='auto',
                 node_set_field_names='auto'):
```

Parameters

`filename`	A filename.
`element_block_ids`	A list of element block ids.
`timesteps`	A list of any number of timesteps.
`side_set_ids`	A list of side set ids.
`node_set_ids`	A list of node set ids.
`global_variable_names`	A list of global variable names.
`node_field_names`	A list of node field names.
`element_field_names`	A list of element field names.
`side_set_field_names`	A list of side set field names.
`node_set_field_names`	A list of node set field names.

Example

```
>>> model.write_model('output.g')
```

export_stl_file

Export the exterior of the model to an STL file.

By default, if timesteps exist and a displacement field exists, the displacements at the last timestep will be applied.

Interface

```
def export_stl_file(self,
                    filename,
                    element_block_ids='all',
                    displacement_timestep='auto'):
```

Parameters

`filename`	A filename.
`element_block_ids`	A list of element block ids.
`displacement_timestep`	The timestep value to use for displacement values.

Example

```
>>> model.export_stl_file('mesh_surface.stl')
```

export_wrl_model

Export the exterior of the model to a WRL file with colors specified by the value of a field.

The WRL file format is used by 3D printing software.

Interface

```
def export_wrl_model(self,
                     filename,
                     node_field_name,
                     element_block_ids='all',
                     timestep='last',
                     field_range='auto',
                     intervals=9,
                     colorspace='rgb',
                     displacement_timestep='auto',
                     export_exodus_copy=True):
```

Parameters

`filename`	A filename.
`node_field_name`	A node field name.
`element_block_ids`	A list of element block ids.
`timestep`	A timestep value
`field_range`	The range of field values.
`intervals`	The number of intervals to use.
`colorspace`	Colorspace to use.
`displacement_timestep`	The timestep value to use for displacement values.
`export_exodus_copy`	If `True`, a copy of the model will also be output to an Exodus file.

Example

```
>>> model.export_wrl_model('colored_eqps_model.wrl', 'eqps')
```

get_element_block_connectivity

Return the connectivity list of an element block.

Interface

```
def get_element_block_connectivity(self, element_block_id='auto'):
```

Parameters

`element_block_id`	An element block id.

Example

```
>>> model.get_element_block_connectivity(1)
```

get_element_block_ids

Return the list of defined element block ids.

Interface

```
def get_element_block_ids(self):
```

No parameters

Example

```
>>> model.get_element_block_ids()
```

get_element_count

Return the total number of elements in the given element blocks.

Interface

```
def get_element_count(self, element_block_ids='all'):
```

Parameters

`element_block_ids` A list of element block ids.

Example

```
>>> print object.get_element_count()
```

get_element_field_names

Return the list of defined element field names.

By default, this returns element fields which are defined on any element block. To return fields which are defined on a particular element block, pass an element block id.

Interface

```
def get_element_field_names(self, element_block_ids='all'):
```

Parameters

`element_block_ids` A list of element block ids.

Example

```
>>> model.get_element_field_names()
>>> model.get_element_field_names(1)
```

get_element_field_values

Return the list of element field information for the given element block, field name, and timestep.

Interface

```
def get_element_field_values(self,
                             element_field_name,
                             element_block_id='auto',
                             timestep='last'):
```

Parameters

`element_field_name`	An element field name.
`element_block_id`	An element block id.
`timestep`	A timestep value

Example

```
>>> model.get_element_field_values('strain', element_block_id=1)
>>> model.get_element_field_values('strain', timestep=2.0)
>>> model.get_element_field_values('strain',
...                                element_block_id=5,
...                                timestep='last')
```

get_global_variable_names

Return the list of global variable names.

Interface

```
def get_global_variable_names(self):
```

No parameters

Example

```
>>> model.get_global_variable_names()
```

get_input_deck

Return each line of the input deck stored in the file.

Many SIERRA applications, when running a problem, will store the input deck within the results file. This function retrieves that information, if it exists. Note that due to format restriction, the retrieved input deck may not exactly match the original file.

Interface

```
def get_input_deck(self):
```

No parameters

Example

```
>>> object = exomerge.import_model('results.e')
>>> model.get_input_deck()
```

get_length_scale

Return the length scale of the model.

The length scale is defined as the largest of the following:

- absolute nodal coordinate component
- total range in nodal coordinate component

Interface

```
def get_length_scale(self):
```

No parameters

Example

```
>>> model.get_length_scale()
```

get_node_field_names

Return the list of defined node field names.

Interface

```
def get_node_field_names(self):
```

No parameters

Example

```
>>> model.get_node_field_names()
```

get_node_field_values

Return the list of node field values for the given field and timestep.

By default, this returns values at the last timestep.

Interface

```
def get_node_field_values(self, node_field_name, timestep='last'):
```

Parameters

`node_field_name`	A node field name.
`timestep`	A timestep value

Example

```
>>> model.get_node_field_values('disp_x')
>>> model.get_node_field_values('disp_x', 0.0)
```

get_node_set_field_names

Return the list of defined node set field names.

By default, this returns node set fields which are defined on any node set. To return fields which are defined on a particular node set, pass a node set id.

Interface

```
def get_node_set_field_names(self, node_set_ids='all'):
```

Parameters

`node_set_ids`	A list of node set ids.

Example

```
>>> model.get_node_set_field_names()
>>> model.get_node_set_field_names(1)
```

get_node_set_field_values

Return the list of node set field information for the given node set, field name, and timestep.

Interface

```
def get_node_set_field_values(self,
                              node_set_field_name,
                              node_set_id='auto',
                              timestep='last'):
```

Parameters

`node_set_field_name`	A node set field name.
`node_set_id`	A node set id.
`timestep`	A timestep value

Example

```
>>> model.get_node_set_field_values('contact_pressure', node_set_id=1)
>>> model.get_node_set_field_values('contact_pressure', timestep=2.0)
>>> model.get_node_set_field_values('contact_pressure',
...                                 node_set_id=5,
...                                 timestep='last')
```

get_node_set_ids

Return the list of defined node set ids.

Interface

```
def get_node_set_ids(self):
```

No parameters

Example

```
>>> model.get_node_set_ids()
```

get_node_set_members

Return the list of node indices that belong to the given node set.

Interface

```
def get_node_set_members(self, node_set_id):
```

Parameters

`node_set_id` A node set id.

Example

```
>>> model.get_node_set_members(1)
```

get_nodes_in_element_block

Return a list of all node indices used in the given element blocks.

Interface

```
def get_nodes_in_element_block(self, element_block_ids):
```

Parameters

`element_block_ids` A list of element block ids.

Example

```
>>> model.get_nodes_in_element_block(1)
>>> model.get_nodes_in_element_block([1, 3])
```

get_nodes_in_side_set

Return a list of node indices which belong to the given side set.

Interface

```
def get_nodes_in_side_set(self, side_set_id):
```

Parameters

`side_set_id` A side set id.

Example

```
>>> model.get_nodes_in_side_set(1)
```

get_side_set_field_names

Return the list of defined side set field names.

By default, this returns side set fields which are defined on any side set. To return fields which are defined on a particular side set, pass a side set id.

Interface

```
def get_side_set_field_names(self, side_set_ids='all'):
```

Parameters

`side_set_ids` A list of side set ids.

Example

```
>>> model.get_side_set_field_names()
>>> model.get_side_set_field_names(1)
```

get_side_set_field_values

Return the list of side set field information for the given side set, field name, and timestep.

Interface

```
def get_side_set_field_values(self,
                              side_set_field_name,
                              side_set_id='auto',
                              timestep='last'):
```

Parameters

`side_set_field_name` A side set field name.

`side_set_id` A side set id.

`timestep` A timestep value

Example

```
>>> model.get_side_set_field_values('contact_pressure', side_set_id=1)
>>> model.get_side_set_field_values('contact_pressure', timestep=2.0)
>>> model.get_side_set_field_values('contact_pressure',
...                                 side_set_id=5,
...                                 timestep='last')
```

get_side_set_ids

Return the list of defined side set ids.

Interface

```
def get_side_set_ids(self):
```

No parameters

Example

```
>>> model.get_side_set_ids()
```

get_timesteps

Return the list of timesteps.

Interface

```
def get_timesteps(self):
```

No parameters

Example

```
>>> model.get_timesteps()
```

global_variable_exists

Return `True` if the given global variable exists.

Interface

```
def global_variable_exists(self, global_variable_name):
```

Parameters

`global_variable_name` A global variable name.

Example

```
>>> model.global_variable_exists('timestep')
```

import_model

Import information (including element blocks, nodes, elements, side sets, and node sets) from the given file.

This will add to the current model in memory, so multiple calls will act to merge files. As a shortcut, one may use the `exomerge.import_model` to create a new model from a file.

```
>>> model = exomerge.import_model('output.e')
```

By default, this will import all information from the given ExodusII results file. To import only part of a mesh file, or to load only a particular timestep, one can use the following options.

```
>>> model.import_model('output.e', element_block_ids=[1, 2])
>>> model.import_model('output.e', timesteps='last')
>>> model.import_model('output.e', node_field_names='disp_*')
```

To import only the mesh without any field information, one can use the following syntax.

```
>>> model.import_model('output.e', timesteps='none')
```

Interface

```
def import_model(self,
                 filename,
                 element_block_ids='all',
                 timesteps='all',
                 node_field_names='all',
                 element_field_names='all',
                 side_set_ids='all',
                 node_set_ids='all',
                 global_variable_names='all',
                 node_set_field_names='all',
                 side_set_field_names='all'):
```

Parameters

`filename`	A filename.
`element_block_ids`	A list of element block ids.
`timesteps`	A list of any number of timesteps.
`node_field_names`	A list of node field names.
`element_field_names`	A list of element field names.
`side_set_ids`	A list of side set ids.
`node_set_ids`	A list of node set ids.
`global_variable_names`	A list of global variable names.
`node_set_field_names`	A list of node set field names.
`side_set_field_names`	A list of side set field names.

Example

```
>>> model.import_model('mesh_file.g')
>>> model.import_model('results_file.e')
```

merge_nodes

Merge nodes that are closer than the given tolerance.

Node fields, node sets and node set fields are updated accordingly.

Interface

```
def merge_nodes(self, relative_tolerance=1e-6, suppress_warnings=False):
```

Parameters

`relative_tolerance` A tolerance value.

`suppress_warnings` If `True`, warning messages will not be output.

Example

```
>>> model.merge_nodes(0)
>>> model.merge_nodes()
```

node_field_exists

Return `True` if the given node field exists.

Interface

```
def node_field_exists(self, node_field_name):
```

Parameters

`node_field_name` A node field name.

Example

```
>>> model.node_field_exists('temperature')
```

node_set_exists

Return `True` if the given node set exists.

Interface

```
def node_set_exists(self, node_set_id):
```

Parameters

`node_set_id` A node set id.

Example

```
>>> model.node_set_exists(1)
```

node_set_field_exists

Return `True` if the given node set field is defined on all of the given node set ids.

Interface

```
def node_set_field_exists(self, node_set_field_name, node_set_ids='all'):
```

Parameters

`node_set_field_name`	A node set field name.
`node_set_ids`	A list of node set ids.

Example

```
>>> model.node_set_field_exists('contact_pressure')
```

output_global_variables

Output global variables in CSV format to a file or standard output.

By default, all nodes are used.

By default, information is output for all timesteps and all global variables. This may be changed by specifying which timesteps and variables are output.

Interface

```
def output_global_variables(self,
                            filename=None,
                            global_variable_names='all',
                            timesteps='all'):
```

Parameters

`filename`	A filename.
`global_variable_names`	A list of global variable names.
`timesteps`	A list of any number of timesteps.

Example

```
>>> model.output_global_variables('variables.csv')
>>> model.output_global_variables('variables.csv', timesteps='last')
```

process_element_fields

Process element field information to create node based fields.

For element fields with 8 integration points, this takes the average.

For element fields with 9 integration points, this takes the first one.

This function is provided as a convenience for post processing elements with multiple integration points.

Interface

```
def process_element_fields(self, element_block_ids='all'):
```

Parameters

`element_block_ids`	A list of element block ids.

Example

```
>>> model.process_element_fields()
```

reflect_element_blocks

Reflect the specified element blocks about the given plane.

Since an element becomes inverted when it is reflected across a plane, this also uninverts the elements.

Interface

```
def reflect_element_blocks(self,
                           element_block_ids,
                           point,
                           normal,
                           check_for_merged_nodes=True,
                           adjust_displacement_field='auto'):
```

Parameters

`element_block_ids`	A list of element block ids.
`point`	A vector describing a point.
`normal`	A vector describing a normal.
`check_for_merged_nodes`	If `True`, check for merged nodes between element blocks to avoid inconsistent output.
`adjust_displacement_field`	If `True`, the displacement field will be updated if it exists.

Example

```
>>> model.reflect_element_blocks(1, [0, 0, 0], [1, 0, 0])
```

rename_element_block

Change an element block id.

Interface

```
def rename_element_block(self, current_element_block_id, new_element_block_id):
```

Parameters

`current_element_block_id`	An element block id.
`new_element_block_id`	An element block id.

Example

```
>>> model.rename_element_block(1, 100)
```

rename_element_field

Rename an element field.

Interface

```
def rename_element_field(self,
                         current_element_field_name,
                         new_element_field_name,
                         element_block_ids='all'):
```

Parameters

`current_element_field_name`	An element field name
`new_element_field_name`	An element field name.
`element_block_ids`	A list of element block ids.

Example

```
>>> model.rename_element_field('p', 'pressure')
```

rename_global_variable

Rename a global variable.

Interface

```
def rename_global_variable(self,
                           current_global_variable_name,
                           new_global_variable_name):
```

Parameters

`current_global_variable_name`	A global variable name.
`new_global_variable_name`	A global variable name.

Example

```
>>> model.rename_global_variable('ke', 'kinetic_energy')
```

rename_node_field

Rename a node field.

Interface

```
def rename_node_field(self, current_node_field_name, new_node_field_name):
```

Parameters

`current_node_field_name`	A node field name.
`new_node_field_name`	A node field name.

Example

```
>>> model.rename_node_field('temp', 'temperature')
```

rename_node_set

Change a node set id.

Interface

```
def rename_node_set(self, current_node_set_id, new_node_set_id):
```

Parameters

`current_node_set_id`	A node set id.
`new_node_set_id`	A node set id.

Example

```
>>> model.rename_node_set(1, 100)
```

rename_node_set_field

Rename a node set field.

Interface

```
def rename_node_set_field(self,
                          current_node_set_field_name,
                          new_node_set_field_name,
                          node_set_ids='all'):
```

Parameters

`current_node_set_field_name`	A node set field name.
`new_node_set_field_name`	A node set field name.
`node_set_ids`	A list of node set ids.

Example

```
>>> model.rename_node_set_field('cp', 'contact_pressure')
```

rename_side_set

Change a side set id.

Interface

```
def rename_side_set(self, current_side_set_id, new_side_set_id):
```

Parameters

`current_side_set_id`	A side set id.
`new_side_set_id`	A side set id.

Example

```
>>> model.rename_side_set(1, 100)
```

rename_side_set_field

Rename a side set field.

Interface

```
def rename_side_set_field(self,
                          current_side_set_field_name,
                          new_side_set_field_name,
                          side_set_ids='all'):
```

Parameters

`current_side_set_field_name`	A side set field name.
`new_side_set_field_name`	A side set field name.
`side_set_ids`	A list of side set ids.

Example

```
>>> model.rename_side_set_field('cp', 'contact_pressure')
```

rotate_element_blocks

Rotate all nodes in the given element blocks by the given amount.

By default, if a displacement field exists, this will also rotate the displacement field.

The rotation axis includes the origin and points in the direction of the `axis` parameter.

Interface

```
def rotate_element_blocks(self,
                          element_block_ids,
                          axis,
                          angle_in_degrees,
                          check_for_merged_nodes=True,
                          adjust_displacement_field='auto'):
```

Parameters

`element_block_ids`	A list of element block ids.
`axis`	A vector describing an axis.
`angle_in_degrees`	An angle given in degrees.
`check_for_merged_nodes`	If `True`, check for merged nodes between element blocks to avoid inconsistent output.
`adjust_displacement_field`	If `True`, the displacement field will be updated if it exists.

Example

```
>>> model.rotate_element_blocks(1, [1, 0, 0], 90)
```

rotate_geometry

Rotate the model about an axis by the given angle.

The rotation axis includes the origin and points in the direction of the `axis` parameter.

Interface

```
def rotate_geometry(self,
                    axis,
                    angle_in_degrees,
                    adjust_displacement_field='auto'):
```

Parameters

`axis`	A vector describing an axis.
`angle_in_degrees`	An angle given in degrees.
`adjust_displacement_field`	If `True`, the displacement field will be updated if it exists.

Example

```
>>> model.rotate_geometry([1, 0, 0], 90)
```

scale_element_blocks

Scale all nodes in the given element blocks by the given amount.

By default, if a displacement field exists, this will also scale the displacement field.

Interface

```
def scale_element_blocks(self,
                         element_block_ids,
                         scale,
                         check_for_merged_nodes=True,
                         adjust_displacement_field='auto'):
```

Parameters

`element_block_ids`	A list of element block ids.
`scale`	The scale factor.
`check_for_merged_nodes`	If `True`, check for merged nodes between element blocks to avoid inconsistent output.
`adjust_displacement_field`	If `True`, the displacement field will be updated if it exists.

Example

```
>>> model.scale_element_blocks(1, 0.0254)
```

scale_geometry

Scale the model by the given factor.

By default, if it exists, the displacement field will also be scaled accordingly.

Interface

```
def scale_geometry(self, scale, adjust_displacement_field='auto'):
```

Parameters

`scale`	The scale factor.
`adjust_displacement_field`	If `True`, the displacement field will be updated if it exists.

Example

```
>>> model.scale_geometry(0.0254)
```

side_set_exists

Return `True` if the given side set exists.

Interface

```
def side_set_exists(self, side_set_id):
```

Parameters

`side_set_id`	A side set id.

Example

```
>>> model.side_set_exists(1)
```

side_set_field_exists

Return `True` if the given side set field is defined on all of the given side set ids.

Interface

```
def side_set_field_exists(self, side_set_field_name, side_set_ids='all'):
```

Parameters

`side_set_field_name`	A side set field name.
`side_set_ids`	A list of side set ids.

Example

```
>>> model.side_set_field_exists('contact_pressure')
```

summarize

Print a summary of the information in the current model.

Interface

```
def summarize(self):
```

No parameters

Example

```
>>> model.summarize()
```

timestep_exists

Return `True` if the given timestep exists.

Interface

```
def timestep_exists(self, timestep):
```

Parameters

`timestep`	A timestep value

Example

```
>>> model.timestep_exists(0.0)
```

to_lowercase

Convert the names of all entities to lowercase.

Interface

```
def to_lowercase(self):
```

No parameters

Example

```
>>> model.to_lowercase()
```

translate_element_blocks

Translate the specified element blocks by by the given amount.

Interface

```
def translate_element_blocks(self,
                             element_block_ids,
                             vector,
                             check_for_merged_nodes=True):
```

Parameters

`element_block_ids`	A list of element block ids.
`vector`	A vector.
`check_for_merged_nodes`	If `True`, check for merged nodes between element blocks to avoid inconsistent output.

Example

```
>>> model.translate_element_blocks(1, [1.0, 2.0, 3.0])
```

translate_geometry

Translate the model by the given vector.

Interface

```
def translate_geometry(self, vector):
```

Parameters

`vector`	A vector.

Example

```
>>> model.translate_geometry([1, 2, 3])
```

unmerge_element_blocks

For elements blocks that share nodes, duplicate these nodes to unmerge the blocks.

Node fields, node sets, and node set fields are updated accordingly.

Interface

```
def unmerge_element_blocks(self, element_block_ids='all'):
```

Parameters

`element_block_ids` A list of element block ids.

Example

```
>>> model.unmerge_element_blocks()
```

Chapter 6

Example input files

This chapter lists some example input files for common operations.

Scale a mesh file

The following script will convert a mesh file given in inches to one given in meters.

```
# import the module
import exomerge
# load the mesh
model = exomerge.import_model('mesh_in_inches.g')
# scale the geometry
model.scale_geometry(0.0254)
# save the model to a new file
model.export_model('mesh_in_meters.e')
```

Calculate and output maximum temperature over time

The following script will find and output the maximum value of the 'temperature' field.

```
# import the module
import exomerge
# load the results file of the relevant element block
model = exomerge.import_model('output_results.e', element_block_ids=1)
# calculate a new field
model.calculate_node_field_maximum('temperature')
# output to a file
model.output_global_variables(filename='max_temp.csv',
                              global_variable_names='temperature_max')
```

Create a cohesive zone from a side set

The following script will insert a layer of cohesive zone elements between elements corresponding to side set 1. The created cohesive zone is element block 2.

```
# import the module
import exomerge
# import the mesh
model = exomerge.import_model('mesh.g')
# create a cohesive zone element block 2 from side set 1
model.convert_side_set_to_cohesive_zone(1, 2)
# save the result
model.export_model('mesh_cohesive.g')
```

Delete an element block

The following script will create a new Exodus file with element block 1 removed.

```
# import the module
import exomerge
# load all results
model = exomerge.import_model('results.e')
# delete an element block
model.delete_element_block(1)
# save the model to a new file
model.export_model('most_results.e')
```

Extract an element block

The following script will extract element block 1 and save it to a new file.

```
# import the module
import exomerge
# load a single element block
model = exomerge.import_model('results.e', element_block_ids=1)
# save the model to a new file
model.export_model('single_block_results.e')
```

Combine two meshes into a single file

The following script will combine two meshes into a single file. Care must be taken to ensure definitions for element block ids, side sets, and node sets do not overlap.

```
# import the module
import exomerge
# load the first mesh
model = exomerge.import_model('first_mesh.g')
# add the second mesh
model.import_model('second_mesh.g')
# save the result
model.export_model('combined_mesh.g')
```

Output global variables over time

The following script will output all global variables into CSV format.

```
# import the module
import exomerge
# load the results
model = exomerge.import_model('output_results.e', element_block_ids='none')
# output all variables
model.output_global_variables('global_variables.csv')
```

Interpolate results between timesteps

The following script will create a new timestep and interpolate field values based on nearby timesteps. By default, the method of interpolation is cubic, although this can be changed. The resulting file will include all of the original timesteps as well as the newly created one.

```
# import the module
import exomerge
# load the results
model = exomerge.import_model('output_results.e')
# create a new timestep at t=0.5 using default (cubic) interpolation
model.create_interpolated_timestep(0.5)
# create a new timestep at t=1.5 using linear interpolation
model.create_interpolated_timestep(0.75, method='linear')
# save the model
model.export('output_results_interpolated.e')
```

Retrieve input deck from a results file

The following script will extract the original input deck from a results file. By default, SIERRA saves the input deck as part of the results file. Because of the limitations of the ExodusII format, the input deck is not always preserved properly.

```
# import the module
import exomerge
# load the results
model = exomerge.import_model('output_results.e')
# save the input deck to a file
with open('original_input_deck.i', 'w') as out:
    out.write(model.get_input_deck())
```

Unmerge element blocks

The following script will unmerge nodes shared between element blocks. It does so by finding shared nodes and duplicating them. To only unmerge specific pairs of element blocks, you can pass a list of ids to the function.

```
# import the module
import exomerge
# load the file
model = exomerge.import_model('merged_mesh.g')
# unmerge nodes between all element blocks
model.unmerge_element_blocks()
# output a new file
model.export_model('unmerged_mesh.g')
```

Merge nearby nodes

The following script will merge nodes which are close to one another. To merge exactly coincident nodes, a value 0 may be passed. Otherwise, a relative tolerance value may be passed to specify how close nodes must be to be merged. This tolerance is relative to the model length scale which may be accessed by the `get_length_scale` function.

```
# import the module
import exomerge
# load the file
model = exomerge.import_model('mesh.g')
# merge nearby nodes
model.merge_nodes()
# output a new mesh file
model = exomerge.import_model('merged_mesh.g')
```

Calculate a new field

Exomerge has a number of function to calculate new fields. The following script will calculate the `temp_f` field assuming the `temp` field exists.

```
# import the module
import exomerge
# load the results
model = exomerge.import_model('results.e')
# calculate temperature in Fahrenheit
model.calculate_node_field('temp_f = (temp - 273.15) * 1.8 + 32')
# output the new file
model.export_model('more_results.e')
```

Note that similar functions exist for calculating global variables, element fields, node set fields and side set fields.

Create an STL file

The following script will create an STL file from the external surface of the model. By default, if a displacement field exists, it will be taken into account.

```
# import the module
import exomerge
# load results
model = exomerge.import_model('results.e')
# create an STL file
model.export_stl_file('results.stl')
```

The STL file format describes a collection of triangles which can be used to describe a volume. This file format is often supported by program which deal with geometries, such as Cubit or SolidWorks.

Extract a timestep

The following script will load a single timestep from the results and save it to a new file. This can be beneficial to cut down on the file size.

```
# import the module
import exomerge
# load the model the last timestep
model = exomerge.import_model('results.e', timesteps='last')
# save the model to a new file
model.export_model('last_timestep_results.e')
```

Extract a mesh

The following script will extract the mesh and save it to a new file. This can be useful if you want to cut down on the size of the results file or wish to extract the original mesh from a results file.

```
# import the module
import exomerge
# load the model without any timestep information
model = exomerge.import_model('results.e', timesteps='none')
# save the model to a new file
model.export_model('results_mesh.g')
```

Create a node set from a side set

The following script will create node set 1 from all of the nodes included in side set 1.

```
# import the module
import exomerge
# load the model without any timestep information
model = exomerge.import_model('mesh_with_side_set.g')
# create a new node set
model.convert_side_set_to_node_set(1, 1)
# save the model to a new file
model.export_model('mesh_with_side_set_and_node_set.g')
```

Create a side set

The following script will create a side set which includes all external element faces on the `Y = 0` plane.

```
# import the module
import exomerge
# load the mesh
model = exomerge.import_model('mesh.g')
# create a new node set
model.convert_side_set_to_node_set(1, 1)
# save the model to a new file
model.export_model('mesh_with_side_set.g')
```

The expression does not have to be linear. For example, to create a side set on the surface of a hemisphere, one may use `'sqrt(X^2 + Y^2 + Z^2) = 1 && Z >= 0'`, if such element faces exists.

Index of API Functions

DISTRIBUTION:

1 MS 0899 Technical Library, 8944 (electronic)


Sandia National Laboratories